Parallelizing a Particle Tracer for Flow Visualization

David A. Lane*
NAS Systems Development Branch
NAS Systems Division
NASA Ames Research Center
M/S T27A-2
Moffett Field, CA 94035

Abstract

Particle tracing involves calculating the trajectory (path) of particles through a flow field. When there are thousands of particles, the calculations may require several minutes. Since each particle trace can be computed independently, particle tracing can be considered to be an embarrassingly parallel application. A particle tracing code called Unsteady Flow Analysis Toolkit (UFAT) was parallelized on the Convex, Cray, and SGI multiprocessor systems. The performance was improved by several factors, depending on the number of processors used.

1 Introduction

Particle tracing is a technique commonly used to visualize flow fields generated from numerical simulations in Computational Fluid Dynamics (CFD). Three types of particle traces are sometimes computed: streamlines, pathlines, and streaklines. A streamline is a curve that is tangent to a vector field at an instant in time. Because streamlines are computed based on one vector field, they are also known as instantaneous streamlines. A pathline is a curve that shows the trajectory of a particle released from

*David A. Lane is employted by Computer Sciences Corporation. This work was performed for NASA under Contract NAS 2-12961.

a fixed location, called the seed location. A streakline is a line joining the positions, at an instant in time, of all particles that have been released from a seed location. Streaklines can be simulated by releasing particles continuously from the seed locations at each time step. Streaklines depict time-varying phenomenon that are not easily shown in instantaneous streamlines. For example: vortex shedding, formation, and separation [1].

In a 3D unsteady (time-dependent) flow simulation, scientists often generate tens of thousands of time steps, and each time step can require tens of megabytes of disk space. Interactive visualization of unsteady flow data sets with disk requirements of this magnitude is very difficult or nearly impossible due to the current hardware constraints. Furthermore, it may take several minutes to compute particle traces at each time step. A particle tracing program called the Unsteady Flow Analysis Toolkit (UFAT) has been developed to compute particle traces from large-scale 3D unsteady flow data sets [4]. UFAT computes particle traces from the specified seed locations. If there are several thousand particles, then it can take several minutes to compute the traces at each time step. To take advantage of multiprocessor systems, UFAT was optimized to perform the particle tracing task in parallel.

This paper describes the code modification made to parallelize UFAT, and the performance results of parallel UFAT are presented. First, the features of UFAT are described in the Section 2. In Section 3, a particle tracing algorithm is described. The code modifications made to parallelize UFAT are discussed in Section 4, and the performance results of parallel UFAT are analyzed in Section 5.

2 Unsteady Flow Analysis Toolkit (UFAT)

UFAT was developed to compute streamlines, pathlines, and streaklines from large-scale unsteady flow fields in curvilinear (structured) grids. Streamlines are computed based on one vector field at a specific time step. Pathlines are simply the trajectories of particles through time. Streaklines are computed by releasing particles from the given seed locations at each time step and tracing all the particles that were released in the previous time steps. The time required to compute streamlines is relatively constant for all time steps since the number of particle traces (the number of seed locations) at each time step is constant. The time required to compute pathlines is also relatively constant for all time steps since the number of particles at each time step is the same or less than the number of seed locations. Some particles may exit the grid domain earlier than other particles; thus, the number of particles can decrease as time elapses. For streaklines, the number of particles at time t is $t \times N_s$, where N_s is the number of seed locations. The computation time for streaklines increases as time elapses because the number of particles increases as time elapses.

UFAT currently provides the following basic features: (1) support single and multi- block curvilinear grids with or without rigid body motion; (2) compute instantaneous streamlines, pathlines, and streaklines; (3) support second-order and fourth-order Runge-Kutta integration schemes with adaptive stepsizing; (4) save particle traces for playback; (5) assign color to particles based on a specified scalar quantity; and (6) provide a save/restore option so that particle tracing can be performed over several run sessions.

3 Particle Tracing Algorithm

The basic problem of particle tracing can be described as follows: given a vector function $\vec{V}(p)$ and the initial position of a particle p, find the trajectory (path) of p through the vector field. The particle is assumed to be massless. The path of p is governed by the following equation:

$$\frac{dp}{dt} = \vec{V}(p).$$

Using the second-order Runge-Kutta integration scheme with adaptive stepsizing, let

$$p^* = p_k + h\vec{V}(p_k),$$

$$p_{k+1} = p_k + h(\vec{V}(p_k) + \vec{V}(p^*))/2, \text{ and}$$

$$k = k + 1,$$

where $h = c/max(\vec{V}(p_k))$, $max()$ is the maximum velocity component of $\vec{V}(p_k)$, and $0 < c \leq 1$. The constant c is used for adaptive stepsizing and it controls the number of steps that p will advance in each grid cell. Adaptive stepsizing is required when the velocity varies rapidly in some regions of the grid. The algorithm described above assumes that the flow is steady (time-independent). For unsteady flow, modifications are required to consider time as a parameter in the integration, see [4].

4 Parallelization

For brevity, this section describes only the code modification made to parallelize the streakline calculation. The code modification made for parallelizing streamline and pathline calculations are very similar. Let N_t represent the number of time steps and N_s represent the number of seed locations. Procedure `Advect()` below computes streaklines by stepping through the given time steps.

```
1.  Procedure Advect()
2.  For $t = 1$ to $N_t$ do
3.      For $s = 1$ to $N_s$ do
4.          For $p = 1$ to $Trace\_Length[s]$ do
5.              Advect_Particle( $particle(s, p)$ )
6.          End for
7.          $Trace\_Length[s] = Trace\_Length[s] + 1$
8.          Let $p'$ = a new particle released from seed $s$
9.          $particle(s, Trace\_Length[s]) = p'$
10.     End for
11.  End for
```

The s loop in the above procedure is performed for each seed. In the p loop, particle p in trace s, denoted by $particle(s, p)$, is advected by calling procedure `Advect_Particle()`, which implements the particle integration scheme described in the previous section. A detailed description of this procedure is given in [4]. In line eight of procedure `Advect()` above, a new particle is released from the given seed location at each time step, and then it is stored in the current trace (line nine). The s loop in procedure `Advect()` was chosen for parallelization because it is the outermost loop of the particle tracing algorithm that can be parallelized without any data dependency. Modifications were also made so that two or more processors will not attempt to update the same global variable simultaneously. The program was written such that each iteration of the s loop can be performed in any order without data dependency.

During particle tracing, UFAT requires the velocity field to advect the particles. The velocity field is currently not given in the input data, and it must be computed. There are several approaches for computing the velocity field. Three possible approaches are: preprocessing, lazy evaluation, and cell caching. The first approach is to preprocess (pre-compute) the velocity at each grid point and store the velocity in a global data area. Although this approach is simple, it is not efficient for grids with several million points. Particle traces often only traverse a small region of the grid. Hence, the velocities at many grid points are not used, and velocity calculations at these grid points are unnecessary. Another approach called 'lazy evaluation' is to compute and store the velocities only when they are used. This approach was suggested in [2] to compute the velocity field. When the velocity at a grid point is needed, the program checks if it has already been computed. If the velocity has not been computed, then it is computed and saved. A flag is set to indicate that the velocity has been computed. An advantage of this approach is that it eliminates unnecessary velocity calculations. A disadvantage is that additional memory is needed to store the velocity and its velocity flag at each grid point. During initialization, the flag at each grid point also needs to be initialized. An alternative approach called cell caching, suggested in [3], is to maintain a local velocity vector that is updated as the particle advances from one cell to another. Each processor will maintain a local velocity vector. This technique is attractive for the parallel version of UFAT because it avoids simultaneous velocity modification.

The following subsections discuss the machine-dependent modifications that were made for parallelizing UFAT on the Convex, Cray, and SGI systems. The discussions refer to procedure `Advect()` described earlier.

4.1 Parallelizing UFAT on the Convex Systems

For parallelization on the Convex systems, a compiler directive was placed before the s loop in procedure `Advect()`:

```
1.  Procedure Advect()
2.  For $t = 1$ to $N_t$ do
        #pragma _CNX force_parallel
3.      For $s = 1$ to $N_s$ do
            .
            .
            .
10.     End for
11. End for
```

The `#pragma _CNX force_parallel` directive forces the compiler to parallelize the s loop regardless of any data dependency [5]. In order to use this directive, the `-O3` flag must be included in the compile option list. On the Convex systems, the FORTRAN subroutines are compiled with the `-re` option, which generates reentrant code and allows the FORTRAN codes to be invoked within a parallel code segment. All SAVE declaration statements in the FORTRAN subroutines that are compiled as reentrant codes are removed to avoid simultaneous data update.

4.2 Parallelizing UFAT on the Cray Systems

On the Cray systems, the s loop was multitasked by placing the `#pragma _CRI taskloop` directive before the loop. All codes executed inside the s loop are collectively referred to as a multitasked code region.

```
1.  Procedure Advect()
2.  For $t = 1$ to $N_t$ do
        #pragma _CRI taskloop private(s) shared(<global variables>)
3.      For $s = 1$ to $N_s$ do
            .
            .
            .
10.     End for
11. End for
```

The `-h task3` flag was used during compile to enable the `#pragma` directive. The multitasking feature in Cray requires that all global variables referenced in a multitasked code region be declared as shared variables in the directive statement [6]. This requires modifying all subroutines referenced in the multitasked code region so that global variables are passed as input arguments. For example: in line five of procedure `Advect()` in Section 4, all global variables required by procedure `Advect_Particle()` are passed as input arguments.

4.3 Parallelizing UFAT on the SGI Systems

Several SGI library functions for parallel programming were used to parallelize UFAT on the SGI systems. The s loop was replaced by the call to `m_fork(Parallel_Advect)`, where `m_fork()` creates a number of processes to execute procedure `Parallel_Advect()` in parallel. The number of processes depends on the number of processors that are available on the system and is set by calling `m_set_procs(n_processors)`. The argument `n_processors` is set to the returned value of `prctl(PR_MAXPPROCS)`, which returns the maximum number of processors that the calling process can use [7]. Procedure `Parallel_Advect()` is shown below:

```
Procedure Parallel_Advect()
while ( (s = m_next()) <= $N_s$) do
      For $p = 1$ to $Trace\_Length[s]$ do
          Advect_Particle( $particle(s, p)$ )
      End for
      $Trace\_Length[s] = Trace\_Length[s] + 1$
      Let $p'$ = a new particle released from seed $s$
      $particle(s, Trace\_Length[s]) = p'$
End while
```

When the `m_fork()` library function is called, it resets a global counter to zero. The counter is incremented each time when `m_next()` is invoked, and it is the number of iterations that have been performed in the parallel loop (the while loop shown above). A requirement of `m_fork()` is that the passing function can have at most six arguments. Procedure `Parallel_Advect()` was written so that all input arguments are declared as global variables instead.

5 Performance Results

Two unsteady flow data sets were used to evaluate the performance of parallel UFAT. Tests were performed on the five systems listed below. Each test was executed several times, and the best results were used. On the SGI and Convex systems, the tests were performed when the systems were mostly idle. It was difficult to find idle time on the Cray C90 system; hence, batch jobs were submitted to run the tests in a time-sharing environment.

- Silicon Graphics 4D/320, two 33 MHZ processors, 256 MB memory
- Silicon Graphics 4D/340, four 33 MHZ processors, 128 MB memory
- Convex C3240, four 25 MHZ processors, 1 GB memory
- Convex C3820, two 66 MHZ processors, 1 GB memory

- Cray C90, sixteen 240 MHZ processors, 256 MW memory

5.1 Delta Wing

The first test data set was generated from a numerical simulation of the descending Delta Wing which consists of 900 thousand grid points. The size of the grid file is 16 megabytes (MB) and the solution file is 20 MB. For the performance tests, 200 time steps were used, which required 200x(16+20) MB = 7.2 gigabytes of disk storage. Since there was not enough disk space on the SGI systems to store the entire test data set, the grid is assumed to be fixed (steady) and 18 solution files were repeated for a total of 200 time steps. At each time step, UFAT released 163 particles into the flow field from the given seed locations to simulate streaklines. Table 1 shows the performance results of UFAT on the SGI 4D/320 and 4D/340 systems. For the performance analysis, the following parameters are reported: (1) execution time, which is measured in wall clock minutes; (2) speedup factor, which is execution time divided by the execution time of the program running on one CPU; (3) efficiency, which is speedup factor divided by the number of CPU's used. Efficiency measures how well the code was parallelized. As shown in Table 1, the performance improved by a factor of 1.3 times when two processors were used on the SGI 4D/320 system. On the SGI 4D/340 system, the performance increased by 2.1 times when four processors were used. The disk I/O time on these systems account for approximately 25% of the execution time. Thus, the efficiency numbers were low (0.63 and 0.53).

	SGI 4D/320			SGI 4D/340		
CPUs	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	108.4 m	1.0	—	75.0 m	1.0	—
2	82.0 m	1.3	0.65	—	—	—
4	—	—	—	35.4 m	2.1	0.53

Table 1. Performance of UFAT on two SGI systems using the Delta Wing data set.

Figure 1 depicts the performance plot of UFAT on the SGI 4D/340 system using one processor. The solid (filled) bar at each time step denotes the I/O time for that time step and the unfilled bar denotes the computation time at each time step. The unfilled bar is stacked on top of the solid bar. Hence, the height of each stacked bar represents the combined I/O and computation time at a specific time step. The disk I/O rate of the SGI 4D/340 system used for the test was approximately 4 MB per second. Figure 2 depicts the performance plot when four processors were used on the SGI 4D/340 system. The I/O time bars in Figures 1 and 2 are almost identical because the parallelization is performed on the computation part of the code only. The performance became I/O bound (i.e. the I/O time accounts for more than half of the execution time) when four processors were used on the SGI 4D/340 system. The performance of the I/O was very unstable on the two SGI systems used. This is evident by the rapid changes in the height of the solid bars shown in Figures 1 and 2. The reason for this unstable I/O rate could be caused by a buffer caching problem on the system.

Table 2 shows the performance results on the Convex C3240 and C3820 systems. The performance improved by a factor of 2.5 times when four processors were used on the Convex C3240. As shown in Table 2, when two processors were used, the performance improved by 1.7 times on the Convex C3820. Figures 3 and 4 depict the performance plots on the Convex C3240 system. The disk I/O rate on the Convex C3240 system used for the test was approximately 30-40 MB per second. From the figures, it is can be seen that the overall performance is no longer I/O bound because of the fast disk I/O rate.

	Convex C3240			Convex C3820		
CPUs	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	68.1 m	1.0	—	31.4 m	1.0	—
2	—	—	—	18.7 m	1.7	0.85
4	27.4 m	2.5	0.63	—	—	—

Table 2. Performance of UFAT on two Convex systems using the Delta Wing data set.

Table 3 shows the performance of UFAT on the Cray C90 system. On the Cray C90, a fast disk (solid-state disk) was used to store the test data set; hence, the I/O time was negligible. When two processors were used, the speedup factor was 2.0 times. Based on the the efficiency numbers shown in the table, it can be concluded that the performance did not improve significantly when eight processors were used. This is due to the fact that the amount of work in the multitasked code region is not significant enough to fully take advantage of eight processors. Figures 5 and 6 depict the performance plots on the Cray C90 using one and two processors. The performance plots for using four and eight processors do not differ significantly from Figure 6, thus they are not shown.

	Cray C90		
CPUs	Time	Speedup	Efficiency
1	15.7 m	1.0	—
2	7.8 m	2.0	1.0
4	5.4 m	2.9	0.73
8	4.4 m	3.6	0.45

Table 3. Performance of UFAT on the Cray C90 using the Delta Wing data set.

5.2 Clipped Delta Wing

The second test data set is a clipped Delta Wing with oscillating control surfaces. Each oscillation cycle consists of 5,000 time steps. For visualization, every 50th time step is saved for a total of 100 time steps per cycle. The clipped Delta Wing grid consists of 250 thousand points in seven blocks. Each grid file is 4 MB and the solution file is 5 MB. Thus, a total of 100x(4+5) MB = 900 MB disk space was used. Two cycles of the data (200 time steps) were used to compute streaklines. At each time step, 36 particles were released from the seed locations. Table 4 depicts the performance results of UFAT on two SGI systems. On the SGI 4D/320 system, the performance improved by a factor of 1.3 times when two processors were used. On the SGI 4D/340 system, the performance improved by 2.1 times when four processors were used. The efficiency numbers are identical to those shown in Table 1. This is due to the slow disk I/O performance on the two SGI systems used.

	SGI 4D/320			SGI 4D/340		
CPUs	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	45.0 m	1.0	—	36.8 m	1.0	—
2	31.0 m	1.3	0.65	—	—	—
4	—	—	—	15.4 m	2.1	0.53

Table 4. Performance of UFAT on two SGI systems using the clipped Delta Wing data set.

Table 5 shows the performance on two Convex systems. On the Convex C3240 system, the performance improved by a factor of 2.9 times when four processors were used. On the Convex C3820 system, the computation time improved by 1.6 times when two processors were used. The efficiency numbers on the two Convex systems are acceptable.

	Convex C3240			Convex C3820		
CPUs	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	32.5 m	1.0	—	16.5 m	1.0	—
2	—	—	—	10.2 m	1.6	0.80
4	11.2 m	2.9	0.73	—	—	—

Table 5. Performance of UFAT on two Convex systems using the clipped Delta Wing data set.

The performance on the Cray C90, as shown in Table 6, improved by factors of 1.9, 3.2, and 4.1 times when two, four, and eight processors were used, respectively. The efficiency numbers shown in the table indicate that the performance did not improve significantly when eight processors were used. This is again due to the insufficient amount of work in the multitasked code region.

	Cray C90		
CPUs	Time	Speedup	Efficiency
1	4.5 m	1.0	—
2	2.4 m	1.9	0.95
4	1.4 m	3.2	0.80
8	1.1 m	4.1	0.51

Table 6. Performance of UFAT on the Cray C90 using the clipped Delta Wing data set.

6 Summary

A particle tracing code was parallelized on the Convex, Cray, and SGI systems. On the Convex systems, parallel UFAT ran reasonably efficient on the Convex C3240 and the Convex C3820, and the performance was no longer I/O bound because of the fast disk I/O rate. On the Cray C90 system, the performance of parallel UFAT did not improve significantly when eight processors were used. This is because the work load in the multitasked code region was not significant enough to benefit from using eight processors. Parallel UFAT did not achieve the expected performance on the SGI systems due to the slow disk I/O rate on the SGI systems used.

Acknowledgments

The author would like to thank David Kenwright, Sam Uselton, and Tom Woodrow for their suggestions. The author also thanks Ed Hamrick for provided many helpful suggestions for parallelizing UFAT on the Convex system.

References

[1] Corrie, I., The Fundamental Mechanics of Fluids, McGraw-Hill, New York, 1974.

[2] Globus, A., A Software Model for Visualization of Time Dependent 3-D Computational Fluid Dynamics Results, *NAS Applied Research Technical Report,* NASA Ames Research Center, RNR 92-031, November 1992.

[3] Hultquist, J., Improving the Performance of Particle Tracing Curvilinear Grids, *32nd AIAA Aerospace Sciences Meeting and Exhibit,* Reno, Nevada, January 1994. AIAA 94-0324.

[4] Lane, D., UFAT - a Particle Tracer for Time-Dependent Flow Fields, in: D. Bergeron and A. Kaufman, eds., *Proceedings of Visualization '94,* Washington, D.C., October 1994, to appear.

[5] CONVEX C Optimization Guide, Second Edition, April 1991.

[6] Cray Standard C Programmer's Reference Manual, SR-2074 3.0, 1991.

[7] m_fork(), Silicon Graphics IRIX Online Manual, release 4.0.3, August 1991.